A Linear-Time Algorithm to Compute the Conjugate of Convex Piecewise Linear-Quadratic Bivariate Functions

Tasnuva Haque, Yves Lucet

January 20, 2018

Abstract

We propose the first algorithm to compute the conjugate of a bivariate Piecewise Linear-Quadratic (PLQ) function in optimal linear worst-case time complexity. The key step is to use a planar graph, called the entity graph, not only to represent the entities (vertex, edge, or face) of the domain of a PLQ function but most importantly to record adjacent entities. We traverse the graph using breadth-first search to compute the conjugate of each entity using graph-matrix calculus, and use the adjacency information to create the output data structure in linear time.

Keywords. Legendre-Fenchel transform; Conjugate; Piecewise linear-quadratic functions; Subdifferential; Convex Function; Computational Convex Analysis (CCA); Computer-Aided Convex Analysis.

1 Introduction

Convex conjugate functions play a significant role in duality theory. Consider the primal optimization problem

$$p = \inf_{x\in\mathbb{R}^d}[f(x) + g(Ax)],$$

whose Fenchel dual optimization problem is

$$d = \sup_{y\in\mathbb{R}^d}[-f^*(A^Ty) - g^*(-y)],$$

where f^* denotes the (Fenchel) conjugate of f, and A^T the transpose of the matrix A. Fenchel's duality Theorem states that under some constraint qualification conditions, $p = d$ and the solution of one problem can be computed from the solution of the other. Understanding how to compute the conjugate of a function is one of the key steps to solve a dual optimization problem [Her16].

Computing the conjugate efficiently is also important for regularization. There is a close relationship [Luc06] between the conjugate and the Moreau envelope (also known as Moreau-Yosida approximate or Moreau-Yosida regularization)

$$M_\lambda(s) = \inf_{x\in\mathbb{R}^d}\left[f(x) + \frac{||x-s||^2}{2\lambda}\right].$$

The Moreau envelope regularizes a nonsmooth function while keeping the same local minima [Luc10, PW16]. The set of point at which the infimum is attained is called the proximal mapping and is the basis of many convex and nonconvex optimization techniques like bundle methods [HUL93].

The importance of conjugate functions inspired us to develop an optimal worst-case time algorithm to compute the entire graph of the conjugate to better understand its structure, and gain

more intuition in the information it carries. Such computation belongs to the field of Computational Convex Analysis (CCA) by contrast with computing the conjugate at a single point, which is an optimization problem.

CCA focuses on creating efficient algorithms to compute transforms commonly encountered in convex analysis: addition, scalar multiplication, Moreau envelope, and (Legendre-Fenchel) conjugate. Those transforms are considered the core convex transforms [GL13, Luc10] and have been used to develop new theories and results in convex optimization [Luc13b]. One application of CCA is Computer-Aided Convex Analysis, which focuses on visualizing transforms applied to low-dimensional functions.

The Computational Convex Analysis (CCA) toolbox has been developed in the past decade to implement the latest algorithms to compute convex transforms efficiently. It is a free and open source Scilab [Con94] toolbox. The CCA toolbox is almost complete for univariate functions but lacking for bivariate functions.

The first algorithm to compute the conjugate of a convex bivariate PLQ function was developed in [GL13] and implemented in Scilab using the Computational Geometry Library CGAL [CGA]. It uses a planar arrangement to store the entities (vertices, edges and faces) of the domain of a PLQ function. The dual arrangement is then computed by looping through the the vertices and computing the conjugate of the associated edges using a decision tree that enumerates all the possibilities. The algorithm uses a binary search tree to detect duplicate points, which results in a $\mathcal{O}(N \log N)$ worst-case time complexity for a function with N pieces. It achieves a linear-time expected case complexity when implemented using a hash table but needs to assume bounded bit length to achieve linear-time worst-case time complexity when implemented using a trie data structure (a try – the name comes from re*trie*val but is pronounced "try" – is a "M-ary tree, whose nodes are M-place vectors with components correspoinding to digits or characters" [Knu73, Section 6.3, p. 492]). Our proposed algorithm is simpler and achieves the same performance without such assumption.

For a convex bivariate PLQ function, the full conjugate can be computed using the partial conjugate, which is the conjugate with respect to only one of the variables. Partial conjugates are also PLQ functions but not necessarily convex [Roc70], e.g. the partial conjugate of the l_1 norm $l_1(x_1, x_2) = |x_1| + |x_2|$ is $(l_1)_1^*(s_1, x_2) = \iota_{[-1,1]}(s_1) - |x_2|$. In [GJL14], a simpler algorithm than [GL13] is proposed to compute the full conjugate of a bivariate PLQ function by using two partial conjugates; both the full and partial conjugate algorithms achieve the same complexity.

Another algorithm, based on parametric programming, was developed in [Jak13] to compute the conjugate of convex bivariate PLQ functions. It combines a Parametric Quadratic programming (pQP) approach with computational convex analysis. The input and output PLQ functions are stored using a list representation which is internally converted into a face-constraints adjacency representation. While a planar graph was used to store the entities of a PLQ function, the adjacency information of the entities was not stored. That algorithm computes the vertices, which requires a log-linear time complexity in the worst-case. The time required to compute the remaining entities, i.e. rays and segments, is log-quadratic because for every vertex, the algorithm loops through all adjacent edges. However, the worst-case time complexity can be improved [Jak13, Theorem 5.7] by using a half-edge data structure to store the vertices in order. Even with this improvement, the algorithm still needs to detect duplicate entities and only achieves a log-linear worst-case time asymptotic complexity.

A summary of the algorithms for computing the conjugate of a bivariate PLQ function is presented in Table 1, which shows that no algorithm is known so far to compute the conjugate of a bivariate PLQ function in linear worst-case time. We present such a new algorithm in the present paper. To our knowledge, it is the first linear-time algorithm. We implement our algorithm in

Table 1: The worst-case time complexity of algorithms developed for computing the conjugate of a bivariate PLQ function. All algorithms achieve linear worst-case space complexity.

Algorithm	Source	Time	Data Structure	Restriction
Full conjugate (geometric algorithm)	[GL13]	Log-linear	Red-Black Tree	
		Linear	Trie	bounded bit length
		Linear	Hash table	Expected time
Full conjugate (parametric optimization)	[Jak13]	Log-linear	List	
Partial conjugate (geometric algorithm)	[GJL14]	Log-linear	Red-Black Tree	

Scilab [Con94].

There are two steps to compute the conjugate: a local step that requires computing dual entities one by one, and a global step that input those dual entities into the output data structure while preventing duplicates. Our contribution is twofold. First, for the local step, we propose a simpler approach that uses graph-matrix calculus [Goe08, GL11] instead of a geometric [GL13, GJL14] or a parametric optimization [Jak13] approach. That improvement makes the algorithm simpler but does not improve the worst-case time complexity. Our second contribution improves the global step by using a neighborhood graph to build the output data structure instead of performing a search using a binary search tree (Red-Black tree), a trie, or a hash table [GL13, GJL14, Jak13]. By using additional information about the conjugate, we are able to avoid any search step completely and achieve an optimal linear worst-case time complexity.

This paper is organized as follows. All the basic notations and definitions required to explain our algorithm are included in Section 2. We include a detailed explanation of the data structure we used in Section 3. Our proposed algorithm is explained with an example in Section 4. The complexity of the algorithm is included in Section 5. Section 6 concludes the paper.

2 Preliminaries and Notations

First, we fix our definitions and notations. We note $\operatorname{ri} C$ (resp. $\operatorname{int} C$, $\operatorname{co} C$) the relative interior (resp. the interior, the convex hull) of a set C, and $\operatorname{dom} f$ the effective domain of a function $f : \mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$ i.e. the set of points where f is finite.

Definition 2.1 (Conjugate function)**.** *Consider a proper convex lower semi-continuous (lsc) function $f : \mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$. The conjugate of f, denoted by f^*, is defined as*

$$f^*(s) = \sup_x (\langle s, x \rangle - f(x)).$$

Definition 2.2 (Polyhedral set)**.** *A polyhedral set C in $\mathbb{R}^d$ is the intersection of finitely many half-spaces and is denoted $C = \{x \in \mathbb{R}^d : {a_i}^T x \leq b_i\}$ where $a_i \in \mathbb{R}^d$, $b_i \in \mathbb{R}, i = 1, \ldots, m$.*

Definition 2.3 (Proper face)**.** *A face of a convex set C is a nonempty subset, $F \subseteq C$ such that if $x_1, x_2 \in C$, and for all θ, $0 < \theta < 1$ we have $\theta x_1 + (1-\theta)x_2 \in F$ then $x_1, x_2 \in F$. A face F that is strictly smaller than C is called a proper face.*

Definition 2.4 (Polyhedral decomposition [PS11, Definition 1]). *The set $\mathcal{C} = \{C_k : k \in \mathcal{K}\}$, where $\mathcal{K}$ is a finite index set, is called a polyhedral decomposition of $\mathcal{D} \subseteq \mathbb{R}^d$ if it satisfies the following conditions*

(i) all of its members C_k are polyhedral sets,

(ii) $\bigcup_{k\in\mathcal{K}} C_k = \mathcal{D}$,

(iii) for all $k \in \mathcal{K}$, $\dim C_k = \dim \mathcal{D}$,

(iv) $\operatorname{ri} C_{k_1} \cap \operatorname{ri} C_{k_2} = \emptyset$, *where $k_1, k_2 \in \mathcal{K}, k_1 \neq k_2$.*

Definition 2.5 (Polyhedral subdivision [PS11, Definition 1]). *The set $\mathcal{C}$ is a polyhedral subdivision if $\mathcal{C}$ is a polyhedral decomposition and the intersection of any two members of $\mathcal{C}$ is either empty or a common proper face of both.*

Definition 2.6 (Piecewise Linear-Quadratic (PLQ) function [RW98]). *A function $f : \mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$ is a Piecewise Linear-Quadratic (PLQ) function if its domain can be represented as the union of finitely many polyhedral sets on each of which the function is either linear or quadratic.*

The domain of a PLQ function can always be decomposed into a polyhedral decomposition and on each set C_k, we note $f(x) = f_k(x)$ if $x \in C_k$, $k \in \mathcal{K}$, where

$$f_k(x) = \frac{1}{2} x^T Q_k x + {q_k}^T x + \alpha_k \tag{1}$$

with Q_k a $d \times d$ symmetric matrix, $q_k \in \mathbb{R}^d$ and $\alpha_k \in \mathbb{R}$. For each piece of the PLQ function f, we associate the function $\tilde{f}_k = f_k + \delta_{C_k}$ where δ_{C_k} is the indicator function defined as

$$\delta_{C_k} = \begin{cases} 0, & \text{if } x \in C_k; \\ \infty, & \text{if } x \notin C_k. \end{cases}$$

Example 2.7. *The l_1 norm function,illustrated in Figure 1, is a PLQ function that will be used throughout the paper as an example. Its domain is partitioned into four polyhedral sets on each of which the function is linear,*

$$l_1(x_1, x_2) = |x_1| + |x_2| = \begin{cases} x_1 + x_2, & \text{if } -x_1 \leq 0, -x_2 \leq 0; \\ -x_1 + x_2, & \text{if } x_1 \leq 0, -x_2 \leq 0; \\ -x_1 - x_2, & \text{if } x_1 \leq 0, x_2 \leq 0; \\ x_1 - x_2, & \text{if } -x_1 \leq 0, x_2 \leq 0. \end{cases}$$

Throughout this paper we assume that the input function is a proper convex lsc bivariate PLQ function whose domain is a polyhedral subdivision.

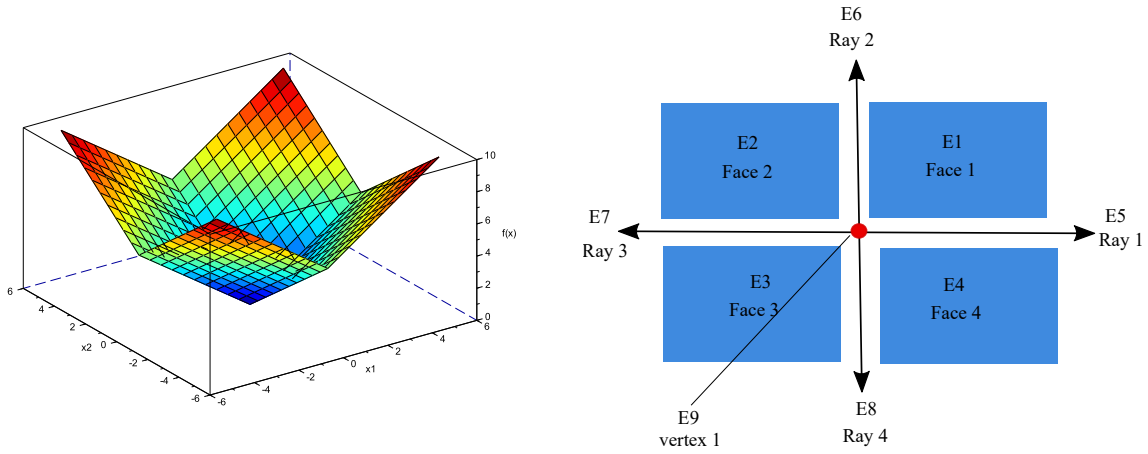
Definition 2.8 (Entity [GL13]). *Assume f is a PLQ function and $\cup_k C_k = \operatorname{dom} f$ is a polyhedral subdivision that induces a partition of $\operatorname{dom} f$. An entity is an element of the partition of the domain. For $f : \mathbb{R}^2 \to \mathbb{R} \cup \{+\infty\}$, an entity is either a vertex, an edge or a face.*

Definition 2.9 (Entity types [Jak13]). *If the dimension of an entity is 0 then it is called a vertex.*

If the dimension of an entity is 1 then it is called an edge and can be written

$$\mathcal{E}_\Lambda = \{x \in \mathbb{R}^d : x = \lambda_1 x_1 + \lambda_2 x_2, \lambda_1 + \lambda_2 = 1, \lambda = (\lambda_1, \lambda_2) \in \Lambda\},$$

where $x_1 \neq x_2$. In $\mathbb{R}^2$, an edge is classified as a



(a) (b)

Figure 1: (a) A bivariate PLQ function - the l_1 norm function (b) Partition of domain of the l_1 norm function.

(i) Line when $\Lambda = \mathbb{R}^2$.

(ii) Ray when $\Lambda = \mathbb{R}_+ \times \mathbb{R}$ *where* $\mathbb{R}_+ = \{x \in \mathbb{R} : x \geq 0\}$.

(iii) Segment when $\Lambda = \mathbb{R}_+ \times \mathbb{R}_+$.

If an entity has a nonempty interior then it is called a face.

Definition 2.10 (Extreme point, [Fan63])**.** *An extreme point of a convex set* C*, is a point* $x \in C$*, such that if* $x = \theta x_1 + (1-\theta)x_2$ *with* $x_1, x_2 \in C$ *and* $\theta \in [0,1]$*, then* $x_1 = x$ *or* $x_2 = x$ *(or both).*

Definition 2.11 (Subgradient, subdifferential [Roc70, RW98])**.** *The subgradient of a function* f *at a point* $\bar{x} \in \operatorname{dom} f$*, is a vector* s *such that*

$$f(x) \geq f(\bar{x}) + \langle s, x - \bar{x} \rangle, \forall x \in \mathbb{R}^d,$$

where $\langle \cdot, \cdot, \rangle$ *denotes the standard dot product.*

The subdifferential of a function f *at a point* $\bar{x}$ *is a closed convex set which is the collection of all subgradients of* f *at* $\bar{x}$*. It is noted*

$$\partial f(x) = \{s \in \mathbb{R}^d : f(x) \geq f(\bar{x}) + \langle s, x - \bar{x} \rangle, \forall x \in \operatorname{dom} f\}.$$

If $x \notin \operatorname{dom} f$ *by convention we set* $\partial f(x) = \emptyset$.

A subgradient of the PLQ function f, defined by Equation (1), at a point x is

$$s = \nabla f_k(x) = Q_k x + q_k. \tag{2}$$

If the function is convex and differentiable at $x \in \operatorname{int} \operatorname{dom} f$, then $\partial f(x) = \{\nabla f(x)\}$. If a PLQ function f is not differentiable at x then its subdifferential is

$$\partial f(x) = \operatorname{co}\{\nabla f_k(x) : \text{ for all } k \text{ such that } x \in C_k\}.$$

Definition 2.12 (Planar graph). *A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is said to be a planar graph if it can be drawn in a plane with no two edges crossing each other except at a vertex to which they are incident.*

The following property states that planar graphs are sparse i.e. they do not have many edges.

Proposition 2.13 ([AMO93, Property 8.7]). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a connected planar graph with n_e edges and n_v vertices. Assume $n_v \geq 3$. Then $n_e \leq 3n_v$.*

We include the proof for completeness. First, we recall Euler's Formula.

Fact 2.14 (Euler's Formula). *For a connected planar graph $\mathcal{G}$ with n_v vertices, n_e edges and n_f faces, we have $n_v - n_e + n_f = 2$.*

Proof. In a planar graph $\mathcal{G}$, the edges divide the plane into different regions and each region is called a face of the graph $\mathcal{G}$. The total number of edges bordering a face F_i is called the degree of F_i. An edge separates 2 faces so $\sum_{i=1}^{m} \deg F_i = 2n_e$.

Since any face has degree $\deg F_i \geq 3$, we obtain $2n_e = \sum_{i=1}^{m} \deg F_i \geq 3n_f$; hence $n_f \leq \frac{2}{3}n_e$. Using Euler's Formula gives $n_f = 2 + n_e - n_v \leq \frac{2}{3}n_e$; so $\frac{1}{3}n_e \leq n_v - 2$, and $n_e \leq 3n_v - 6$. □

Since graph-matrix calculus is a fundamental part of our algorithm, we recall the following fact. We note

$$\operatorname{gph} \partial f = \{(x, s) \in \mathbb{R}^d \times \mathbb{R}^d \ : \ s \in \partial f(x)\}$$

the graph of the subdifferential and Id the identity matrix.

Fact 2.15 (Goebel's Graph-matrix calculus [Goe08, GL11]). *Assume $f : \mathbb{R}^d \to \mathbb{R} \cup \{+\infty\}$ is a proper lsc convex function, then*

$$\operatorname{gph} \partial(f^*) = \begin{bmatrix} 0 & \mathrm{Id} \\ \mathrm{Id} & 0 \end{bmatrix} \operatorname{gph} \partial f.$$

In other words, one can compute the graph of the subdifferential of the conjugate by applying a linear transform to the graph of the subdifferential of the function. We will use the result to associate a dual point (x, s, y^*) to any primal point (x, s, y) with $y^* = s^T x - y$. Note that the result is *vectorized*, i.e. a program like MATLAB can compute the result using vector operations resulting in increased efficiency on any modern computer.

3 Representation of a PLQ function

The input for our algorithm is a graph, called the entity graph, that stores specific information on each entity. The entity graph is built from a proper convex lsc PLQ function. Each node of the entity graph represents an entity of the PLQ function.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the entity graph where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. Each node represents an entity. When two entities are adjacent, we connect them using an edge. For example, the domain of the l_1 norm function is partitioned into four faces, four rays and one vertex, which is illustrated in Figure 1 while Figure 2 is the corresponding entity graph. In addition, each node of $\mathcal{G}$ stores the following information about its entity: the GPH matrix, the adjacent entities and the entity type.

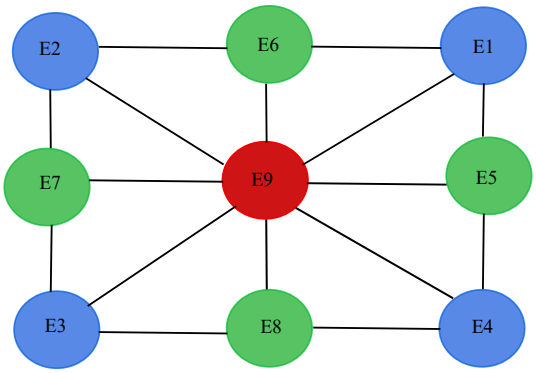


Figure 2: Entity graph for the l_1 norm function

3.1 GPH matrix

We use a GPH matrix to represent an entity C_k and its associated function f_k [Luc13b]. For each entity k, we select some points from C_k and compute the full subdifferential and the function value at those points. Points in the GPH matrix representation are stored in order. Suppose we pick n points to completely represent C_k. Its GPH matrix G_k is

$$G_k = \begin{bmatrix} x_1 & x_2 & \dots & x_n \\ s_1 & s_2 & \dots & s_n \\ y_1 & y_2 & \dots & y_n \\ b_1 & b_2 & \dots & b_n \\ b_1^* & b_2^* & \dots & b_n^* \end{bmatrix}$$

where $x_i \in C_k$ is the coordinate of a point, s_i is a subgradient of f at the point x_i, and y_i is the value of f at x_i. The points x_i represent the polyhedral set C_k and are given in clockwise order. Similarly, the points s_i describe the full subgradient $\partial f(x_i)$, which is a polyhedral set. The binary flag b_i (resp. b_i^*) is used to identify whether x_i (resp. s_i) is an extreme point. It equals 0 for an extreme point and 1 otherwise.

In $\mathbb{R}^d$, the dimension of the GPH matrix is $(2d+3)n$, where n is the number of points. For example, in $\mathbb{R}^2$, the dimension of a GPH matrix is $7n$.

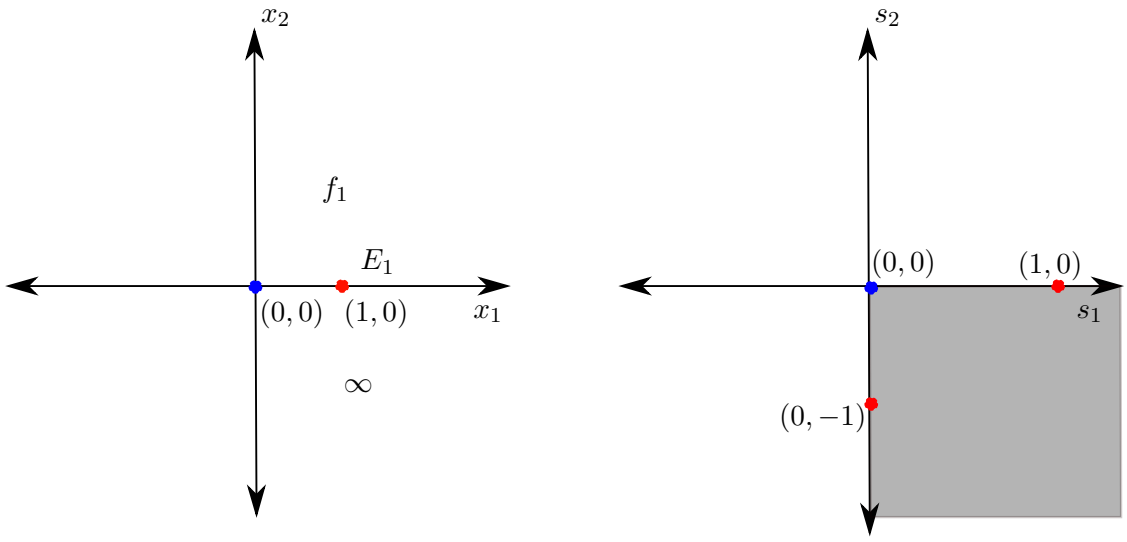
Example 3.1. *Consider* $f(x_1, x_2) = (x_1^2 + x_2^2)/2$ *if* $x_1 \geq 0, x_2 \geq 0$ *and* ∞ *otherwise, whose conjugate is*

$$f^*(s_1, s_2) = \begin{cases} \frac{1}{2}(s_1^2 + s_2^2), & \text{if } s_1 \geq 0, s_2 \geq 0; \\ \frac{1}{2}s_2^2, & \text{if } s_1 \leq 0, s_2 \geq 0; \\ 0, & \text{if } s_1 \leq 0, s_2 \leq 0; \\ \frac{1}{2}s_1^2, & \text{if } s_1 \geq 0, s_2 \leq 0. \end{cases}$$

Hence, the domain of the conjugate is composed of 9 entities: 4 faces, 4 rays, and 1 vertex. The domain of the PLQ function f and the domain of its conjugate f^* are shown in Figure 3.

To compute the GPH matrix for the entity $E_1 = \{(x_1, 0) : x_1 \geq 0\}$*, a ray, we note that* E_1 *is adjacent to* $F_1 = \{x : x_1, x_2 \geq 0\}$ *(associated with* $f_1(x_1, x_2) = \frac{1}{2}{x_1}^2 + \frac{1}{2}{x_2}^2$*) and entity* $F_2 = \{x : x_1 \geq 0, x_2 \leq 0\}$ *(associated with* $f_2(x_1, x_2) = \infty$*). Since* $\operatorname{dom} f_2 = \emptyset$*,* $\partial f_2(x_1, x_2) = \emptyset$*.*

We note that for any point $x \in E_1$*,* $\partial \tilde{f}_1(x) = \{0\} \times (-\infty, 0]$*. Hence, we need 2 points to store* E_1 *(we have to select* $(0, 0)$ *since it is an extreme point and we arbitrarily pick* $(1, 0)$ *to represent the*



(a) Primal graph. The ray $E_1 = \{(x_1, 0) : x_1 \geq 0\}$ is represented using extreme point $(0,0)$ and the arbitrarily chosen nonextreme point $(1,0)$. It separates region F_1 associated with function f_1 from region F_2 associated with $f_2 = +\infty$.

(b) Dual graph. The gray area represents $\partial f(E_1) = \cup_{x \in E_1} \partial f(x)$ where $\partial f(x) = \{0\} \times (-\infty, 0]$; it is represented using the extreme point $(0,0)$ and the (arbitrarily chosen) nonextreme points $(1,0)$ and $(0,-1)$.

Figure 3: The domain of Function f from Example 3.1 (left) and the domain of its conjugate (right). Blue dots represent extreme points while red dots represent non-extreme points.

direction) but 3 points to store its associated entity $\mathbb{R}^+ \times \mathbb{R}^-$ *(again we have to select the extreme point* $(0,0)$ *and arbitrarily pick* $(0,-1)$ *and* $(1,0)$ *to represent the 2 directions). Consequently, a GPH matrix* G_1 *for* E_1 *is*

$$G_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \\ 1/2 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

We note that storing extreme points and directions is sufficient to represent any polyhedral set.

Fact 3.2. *(Representation of a polyhedral set [Roc70, Theorem 19.1, Corollary 19.1.1]) Assume* $C = \{x \in \mathbb{R}^2 : a_i^T x \leq b_i, i = 1, \ldots, m\}$ *is a polyhedral set with at least one extreme point. Then the set of extreme points of* C *contains a finite number of elements* $x_1, x_2, \cdots, x_k$*. If* C *is bounded then the set of extreme direction is empty. If* C *is not bounded then the set of extreme directions is nonempty and has a finite number of elements* $d_1, d_2, \ldots, d_l$*. Moreover,* $\bar{x} \in C$ *if and only if*

$$\bar{x} = \sum_{i=1}^{k} \lambda_i x_i + \sum_{j=1}^{l} \mu_j d_j,$$

where $\sum_{i=1}^{k} \lambda_i = 1, \lambda_i \geq 0, i = 1, \ldots, k, \mu_j \geq 0, j = 1, \ldots, l.$

After we compute the conjugate f^* and provided it with our data structure, the user may wish to compute the quadratic formula associated with each entity; such computation can be performed by interpolation as follow. Given the GPH matrix G_k, we compute the entity C_k and the associated function f_k by solving a linear system amounting to Hermite interpolation. Recall that, on each piece of the domain the function f is quadratic and represented by Equation (1). The subgradient of f at a point x is represented by Equation (2). For each x we get d equations from (2)(gradient information) and 1 equation from knowing the function value. So for each x we have a total of $d+1$ equations.

The number of unknowns to determine the symmetric matrix Q_k is $d+d-1+\cdots+1 = d(d+1)/2$, while it is d for q_k and 1 for α_k giving a total of $d(d+1)/2 + d + 1 = (d+1)d/2 + 1$ unknowns. Consequently, the minimum number of points required to determine the function f_k is $(\lceil d/2 \rceil + 1)$.

In $\mathbb{R}^2$, we need to determine 6 unknowns. For each point we get 3 equations. So we need to store at least two points to determine f_k uniquely, although we may need to store more to represent C_k.

Example 3.3. *Note*

$$f_k(x) = y = \frac{1}{2}\begin{bmatrix} x_1 & x_2 \end{bmatrix}\begin{bmatrix} a & b \\ b & c \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} q_1 & q_2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \alpha.$$

Equation (2) gives

$$s_1 = ax_1 + bx_2 + q_1, \tag{3}$$

$$s_2 = bx_1 + cx_2 + q_2. \tag{4}$$

Consider the entity E_1 of Example 2.7, which is a face. Its GPH matrix G_1 is

$$G_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

We need 3 points to represent E_1. The matrix G_1 indicates that $(0,0)$ is an extreme point while $(1,0)$ and $(0,1)$ are non extreme points.

To determine f_1, the function associated with E_1, we need to compute (a,b,c,q_1,q_2,α). From Equation (3) we solve

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}\begin{bmatrix} a \\ b \\ q_1 \end{bmatrix}.$$

to obtain $(a,b,q_1) = (0,0,1)$. Similarly using (4) we compute $(b,c,q_2) = (0,0,1)$ and substituting we get $\alpha = 0$. We deduce $f_1(x_1,x_2) = x_1 + x_2$.

In the GPH matrix representation, we store a point x multiple times when its subdifferential is multi-valued [GL11, GL10].

Example 3.4. *Consider the entity E_5 of Example 2.7. It is a ray adjacent to two faces: E_1 and E_4. Consider $x = (0,0)$. The face E_1 is associated with $f_1(x) = x_1 + x_2$ and the only subgradient of f_1 at $x = (0,0)$ is $(1,1)$. Similarly, the face E_4 is associated with $f_4(x) = x_1 - x_2$, and at*

$x = (0,0)$, the subgradient is $(1,-1)$. At any point on E_5 (except $(0,0)$) the subdifferential is $\partial f(x_1,x_2) = \{1\} \times [1,-1] = \text{co}\{(1,1),(1,-1)\}$. It has two extreme points $(1,1)$ and $(1,-1)$. We represents the GPH matrix G_5 associated with E_5 as

$$G_5 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Remark 3.5. *In $\mathbb{R}^2$, we can recover the polyhedral set by computing the convex hull of x_i where $i = 1, \cdots, n$ (s_i in the dual). Note that we need at least one extreme point to recover the polyhedral set, so we may split* dom f *further to make sure any entity C_k has at least one extreme point.*

For our algorithm, we choose to store the extreme points in order as we move along the convex hull in clockwise fashion. Since there are either 0 (bounded set), 1 (ray), or 2 (face) directions, we store nonextreme points for unbounded sets as the first and last column.

In our Scilab implementation, we use two hypermatrices H_p and H_d to store the GPH matrices of all entities in the domain of f and the domain of f^* respectively. In $\mathbb{R}^d$, the dimension of the hypermatrix is $N \times M \times n_{\max}$ where N is the total number of entities, $M = (2d+3)$ is the number of rows of the GPH matrix and $n_{\max} = \max\{n_i : i = 1, \ldots, N\}$ where n_i is the number of columns in the GPH matrix of entity i.

Example 3.6. *The function l_1 norm has nine entities and the maximum value of n_i is 4 which is attained for the vertex E_9. So the dimension of the hypermatrix H_p used to store the entities of the l_1 norm is $9 \times 7 \times 4$.*

3.2 Adjacent entities

Each node of $\mathcal{G}$ stores its adjacent entities information in a sparse $N \times m$ matrix, called the neighbor matrix, where N is the number of entities and m is the maximum degree of all vertices in the entity graph.

Example 3.7. *Consider Example 2.7, which has 9 entities. The vertex E_9 has eight adjacent entities which is the maximum. So we use a Neighbor Matrix of dimension 9×8. The entity E_1 is adjacent to E_5, E_6 and E_9. In the entity graph $\mathcal{G}$, the node $\mathcal{V}_1$ which contains E_1, stores the indices of the adjacent entities i.e. 5, 6, 9. We represent the Neighbor Matrix of Example 2.7 as*

$$N_M = \begin{bmatrix} 5 & 6 & 9 & 0 & 0 & 0 & 0 & 0 \\ 6 & 7 & 9 & 0 & 0 & 0 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 & 0 & 0 & 0 \\ 5 & 8 & 9 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 9 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 9 & 0 & 0 & 0 & 0 & 0 \\ 2 & 3 & 9 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 9 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}.$$

Each row of this matrix contains the adjacency information of the corresponding entity. For example, the index of all adjacent entities of E_5 is found in the 5^{th} row of N_M.

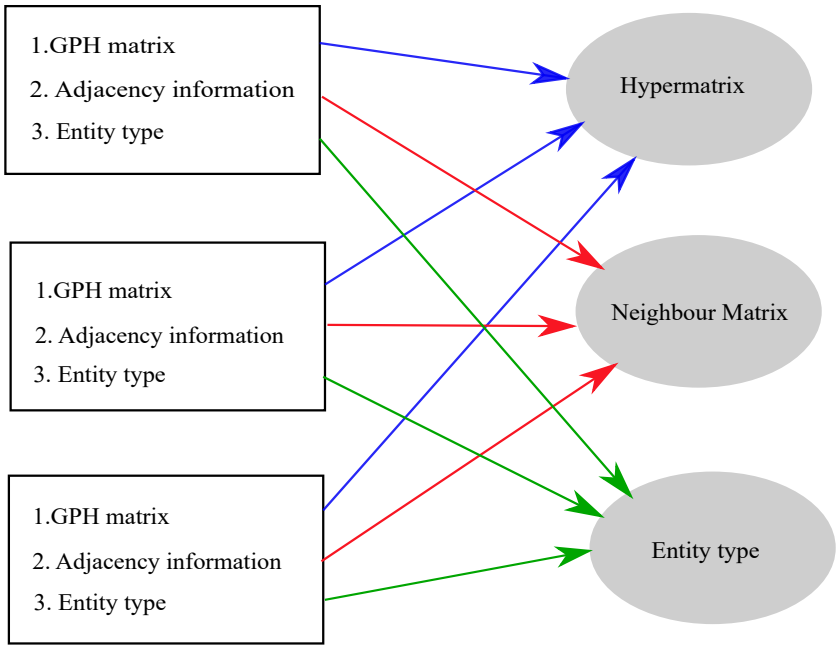


Figure 4: Visualization of the relationship between the information stored for each entity and the data structure recording that information.

3.3 Entity Type

The type of each entity is stored as a single integer whose meaning is listed in Table 2. We use an $1 \times N$ array T to store the type of all entities.

Table 2: Entity type.

Entity Type	Flag
Vertex	1
Face	2
Line	3
Ray	4
Segment	5

Example 3.8. *The array which stores the entity type of the l_1 norm function is*

$$T = \begin{bmatrix} 2 & 2 & 2 & 2 & 4 & 4 & 4 & 4 & 1 \end{bmatrix}.$$

It indicates that entities E_1 to E_4 are faces, entities E_5 to E_8 are rays and E_9 is a vertex.

We visualize our data structure in Figure 4. Consider a PLQ function with N entities where each entity has a GPH matrix, adjacency information and entity type. We store all GPH matrices in a hypermatrix, all adjacency information in a neighbor matrix N_M and all entity types in an array T.

4 Algorithm for computing the conjugate of a PLQ function

Consider $f : \mathbb{R}^2 \to \mathbb{R} \cup \{+\infty\}$ where dom f has N entities. Algorithm 1 uses breadth-first search to traverse the entity graph $\mathcal{G}$. When Entity i is considered, we store the index of all adjacent entities in an array denoted D of dimension $1 \times N$. We traverse $\mathcal{G}$ according to the index stored in D. Note that we will not store any duplicate index in D. To check whether the index of an entity is already stored in D, we use a binary array I of dimension $1 \times N$ ($I_i = 1$ if entity i is already stored in D, otherwise $I_i = 0$).

Algorithm 1 calls two subroutines: Algorithm 2 and Algorithm 3. Algorithm 2 computes the dual entity G_d and type t of the primal entity G_p. It uses graph-matrix calculus on Line 6 to obtain G_d while the remainder of the algorithm computes t. Algorithm 3 updates D, I, $\bar{N}$ using information from E_{adj}. It uses the binary array I to store unique indices in D and allows the main loop of Algorithm 1 to pick the next non-visited index at Line 5 in Algorithm 1.

Algorithm 1 Computing the conjugate of a PLQ function in linear time

Require: Primal information: H_p (a hypermatrix which contains the GPH matrices of all entities), N_M^p (contains the adjacency information of all entities), T_p (contains the type of all entities).

1: **function** COMPUTE_PLQ_CONJUGATE(H_p, N_M^p, T_p)
2: Initialize I with zero and set $I(1) = 1$
3: Initialize D with zero and set $D(1) = 1$ and $\bar{N} = 1$
4: **for** $i \leftarrow 1$ to N **do**
5: $j \leftarrow D$(i)
6: $G_p \leftarrow H_p(j,:,:)$
7: $[G_d, t] \leftarrow$ Conjugate_GPH(G_p)
8: $H_d(j,:,:) \leftarrow G_d$
9: $T_d(j) \leftarrow$ t
10: **if** ($\bar{N} < N$) **then**
11: $E_{adj} = N_M^p(j,:)$
12: Compute_Index($E_{adj}, D, I, \bar{N}$)
13: **end if**
14: **end for**
15: $N_M^d \leftarrow N_M^p$
16: **return** Dual information: H_d, N_M^d, T_d
17: **end function**

Example 4.1. *Applying Algorithm 1 to the l_1 norm function whose entity graph is illustrated on Figure 2, we start from the face E_1 with GPH matrix*

$$G_1^p = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix},$$

and adjacent entities $5, 6, 9$. We initialize $D = [1, 0, 0, 0, 0, 0, 0, 0]$ and $I = [1, 0, 0, 0, 0, 0, 0, 0]$, and

Algorithm 2 Computing the GPH matrix and the type of an entity

Require: G_p (the GPH matrix in the primal).

function CONJUGATE_GPH(G_p)
  // Compute G_d
  $x = G_p(1:2,:)$, $b = G_p(6,:)$
  $y = G_p(5,:)$,
  $s = G_p(3:4,:)$, $b* = G_p(7,:)$
  The conjugate is $G_d = \begin{bmatrix} s \\ x \\ s^T x - y \\ b \\ b^* \end{bmatrix}$
  // Compute t
  P_u = number of unique columns in $\{G_d(1:2,k) : k\}$
  P_t = number of columns of G_d
  if $P_u = 1$ and $P_t \geq 1$ **then**
    // G_d is a vertex
    t = 1
  else if $P_u = 2$ **then**
    //G_d is an edge
    P_{b_0} = number of elements in $\{G_d(6,k) : G_d(6,k) = 0\}$
    P_{b_1} = number of elements in $\{G_d(6,k) : G_d(6,k) = 1\}$
    if $P_{b_1} = 2$ **then**
      // G_d is a line
      t = 3
    else if ($P_{b_0} = 1$ and $P_{b_1} = 1$) **then**
      // G_d is a ray
      t = 4
    else if $P_{b_0} = 2$ **then**
      // G_d is a segment
      t = 5
    end if
  else if $P_u \geq 2$ **then**
    // G_d is a face
    t = 2
  end if
  return G_d, t
end function

$\bar{N} = 1$. *We apply Algorithm 2 to compute the conjugate of* E_1 *and obtain*

$$G_1^d = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}.$$

Since G_1^d *has a unique subgradient, we deduce it is a vertex and set its type to* 1.

Next we check the adjacent entities of E_1 *using Algorithm 3.*

Algorithm 3 Computing the index of adjacent entities

Require: E_{adj} (contains the indices of all adjacent entities of an entity), D (contains the indices of the entities to traverse), I (indicates which entities are already included in D), $\bar{N}$ (number of nonzero elements in D).

```
function COMPUTE_INDEX(E_adj, D, I, N̄)
    Adj ← [extract the non zero elements from E_adj]
    for i ← 1 to size_of(Adj) do
        index ← Adj(i)
        if (I(index) == 0) then
            I(index) = 1
            N̄ ← N̄ + 1
            D(N̄) = index
        end if
    end for
    return D, I, N̄
end function
```

We find that the entity E_1 *is adjacent to the entities* E_5, E_6, *and* E_9. *Then we check the corresponding index of* I *and see that these entities have not been included in* D *yet. So we update* D, I *and* $\bar{N}$ *as*

$$\begin{aligned} I &= \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}, \\ D &= \begin{bmatrix} 1 & 5 & 6 & 9 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \\ \bar{N} &= 4, \end{aligned}$$

and loop to the next entity indicated in D, *in this case Entity* 5.

We traverse all nodes of $\mathcal{G}$ *and update the array* D *and* I *accordingly, see Table 3.*

Figure 5 shows the partition of the primal and the dual domain of the l_1 *norm function. The mapping of the entities from the primal to the dual domain is presented in Table 4.*

5 Complexity analysis

The space and time complexity of our algorithm is computed next.

Table 3: Iterations of Algorithm 1.

Iterations	D									I								
Initialization	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	1	5	6	9	0	0	0	0	0	1	0	0	0	1	1	0	0	1
2	1	5	6	9	4	0	0	0	0	1	0	0	1	1	1	0	0	1
3	1	5	6	9	4	2	0	0	0	1	1	0	1	1	1	0	0	1
4	1	5	6	9	4	2	3	7	8	1	1	1	1	1	1	1	1	1
5	1	5	6	9	4	2	3	7	8	1	1	1	1	1	1	1	1	1
6	1	5	6	9	4	2	3	7	8	1	1	1	1	1	1	1	1	1
7	1	5	6	9	4	2	3	7	8	1	1	1	1	1	1	1	1	1
8	1	5	6	9	4	2	3	7	8	1	1	1	1	1	1	1	1	1
9	1	5	6	9	4	2	3	7	8	1	1	1	1	1	1	1	1	1

Table 4: Mapping of the primal entity to the dual entity for the l_1 norm function.

Primal entity	Type	Dual entity	Type
E_1	face 1	$E_1{}'$	vertex 1
E_2	face 2	$E_2{}'$	vertex 2
E_3	face 3	$E_3{}'$	vertex 3
E_4	face 4	$E_4{}'$	vertex 4
E_5	ray 1	$E_5{}'$	segment 1
E_6	ray 2	$E_6{}'$	segment 2
E_7	ray 3	$E_7{}'$	segment 3
E_8	ray 4	$E_8{}'$	segment 4
E_9	vertex 1	$E_9{}'$	face 1

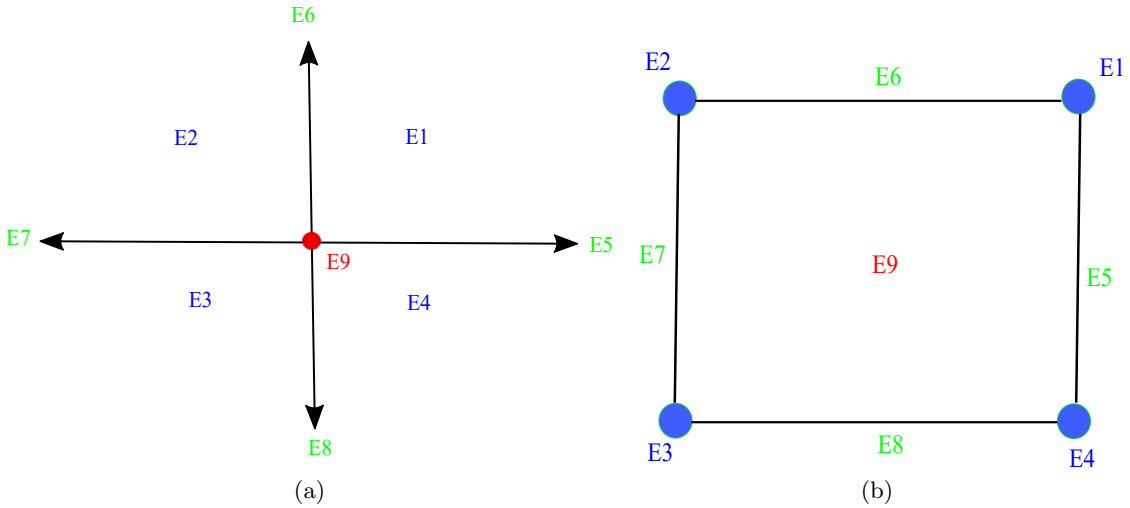


Figure 5: (a) Partition of dom f (b) Partition of dom f^*.

5.1 Space complexity

Proposition 5.1. *Consider a proper convex lsc bivariate PLQ function $f : \mathbb{R}^2 \to \mathbb{R} \cup \{+\infty\}$ with N entities. The worst-case space complexity for computing the conjugate of f using Algorithm 1 is $\mathcal{O}(N^2)$ but can be improved to $\mathcal{O}(N)$ using a sparse matrix data structure.*

Proof. In Algorithm 1, we use two hypermatrices (H_p and H_d) to store the input and output GPH matrices, a Neighbor matrix N_M to store the adjacency information, two arrays (T_p and T_d) to store the type of entities, an array D to store the indices to traverse and a binary array I.

The space complexity of our algorithm mainly depends on the size of the hypermatrix and the Neighbor matrix. Recall that the size of the hypermatrix H is $N \times M \times n_{max}$ where N is the total number of entities, $M = (2d + 3)$ is the number of rows of the GPH matrix and n_{max} is the maximum number of columns in the GPH matrix. If we assume d is constant, the worst-case space complexity of H is $\mathcal{O}(N)$.

In addition, we use four arrays: T_p, T_d, D and I. In the worst-case, the space required to store each array is $\mathcal{O}(N)$.

In Algorithm 1, we use a planar graph as the entity graph. Recall that the sum of the degrees of the vertices equals twice the number of edges and the dimension of the Neighbor matrix N_M is $N \times m$ where m is the maximum degree of all vertices. For N entities, in the worst-case the maximum degree of the vertices is $(N - 1)$. So the worst-case space complexity of N_M is $\mathcal{O}(N^2)$.

Finally, using a sparse matrix data structure we only store the nonzero elements of N_M, which according to Proposition 2.13, amounts to $\mathcal{O}(N)$. □

Remark 5.2. *An adjacency list data structure would also result in a $\mathcal{O}(N)$ space complexity; the quadratic space complexity of our implementation is a Scilab limitation.*

5.2 Time complexity

Proposition 5.3. *The worst-case time complexity for computing the conjugate of a proper convex lsc bivariate PLQ function f with N entities using Algorithm 1 is $\mathcal{O}(N)$.*

Proof. The time complexity of our algorithm comes from the following parts

(i) Extracting N GPH matrices from the hypermatrix H_p according to the index stored in D and storing them in E,

(ii) Computing the conjugate of E,

(iii) Checking the adjacency information of E and updating the arrays D and I,

(iv) Storing the GPH matrix containing the conjugate of E in H_d.

Task (i) amounts to accessing an index in hypermatrix H_p, which takes $\mathcal{O}(1)$; the total time required to access N elements of H_p is $\mathcal{O}(N)$. Task (ii) involves computing the conjugate and takes $\mathcal{O}(N)$. Task (iii) involves storing the index of adjacent entities in the array D. Recall that the indices stored in D are used to traverse the entity graph. In this part, when storing an index in D we need to check whether the index is already stored in D or not. In the worse-case, searching the index of an entity in an array takes $\mathcal{O}(N)$ time. For N entities, the time for storing all indices in D would then be $\mathcal{O}(N^2)$. However, we use a binary array I to determine whether an index is already stored in D. Accessing a position of I takes constant time; so the time complexity for N entities reduces to $\mathcal{O}(N)$. Finally, task (iv) requires accessing a position of hypermatrix H_d and storing the output GPH matrix in H_d, which is performed in linear time like Task (i). Consequently, the overall time complexity for Algorithm 1 is linear. □

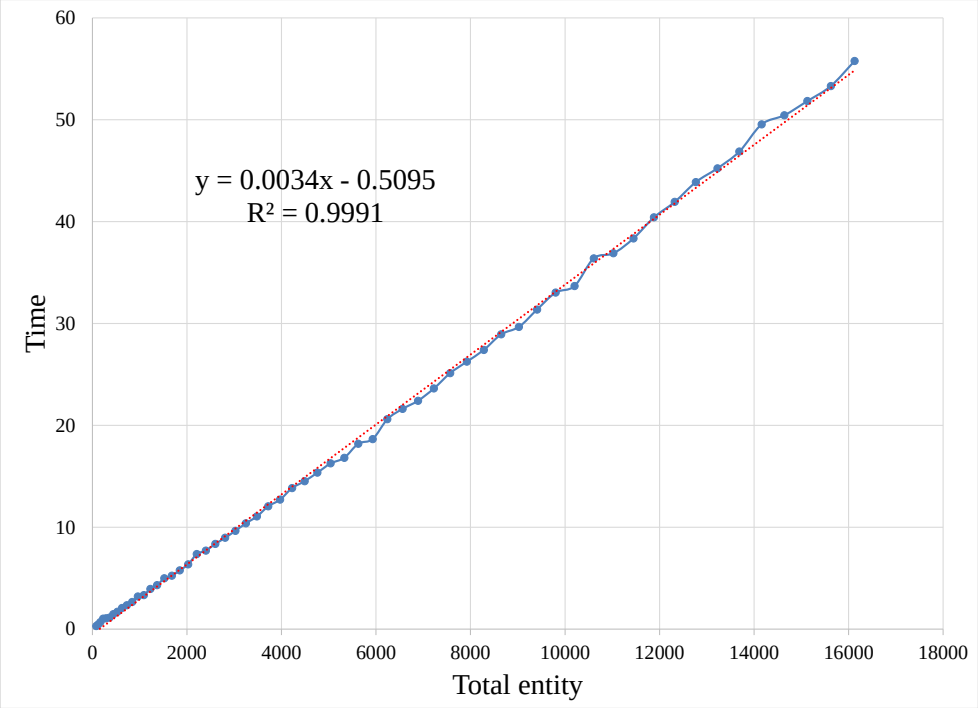


Figure 6: The time complexity for a function approximating (5) with a domain partitioned into a grid.

5.3 Performance Comparison

Consider the following additively separable PLQ function

$$f(x_1, x_2) = x_1^4 + x_2^4. \tag{5}$$

We set $f_1(x_1) = x_1^4$ and $f_2(x_2) = x_2^4$. We use the *plq_build* function from the CCA package [Luc13a] to compute a quadratic approximation of f_1 with an univariate PLQ function. Then we combine the approximation to obtain an approximation of the bivariate function f.

Next we compute all the entities and represents them using GPH matrices. We build the hypermatrix H_p and compute the neighbor matrix N_M. Then we apply our algorithm and compute an approximation of

$$f^*(s_1, s_2) = f_1^*(s_1) + f_2^*(s_2),$$

where $f_1^*(s_1) = (\frac{3}{4})^{\frac{4}{3}} {s_1}^{\frac{4}{3}}$ and $f_2^* = f_1^*$. We increase the number of pieces by increasing the size of the grid and measure the time for computing the conjugate of f. Figure 6 illustrates the linear time complexity of our algorithm.

We compare the performance of our proposed algorithm with the algorithm developed in [Jak13] on Figure 7. For the algorithm developed in [Jak13], a linear least-square regression results in $R^2 = 0.91$ vs. $R^2 = 0.99$ for a quadratic regression i.e. the implementation tested on the specific example

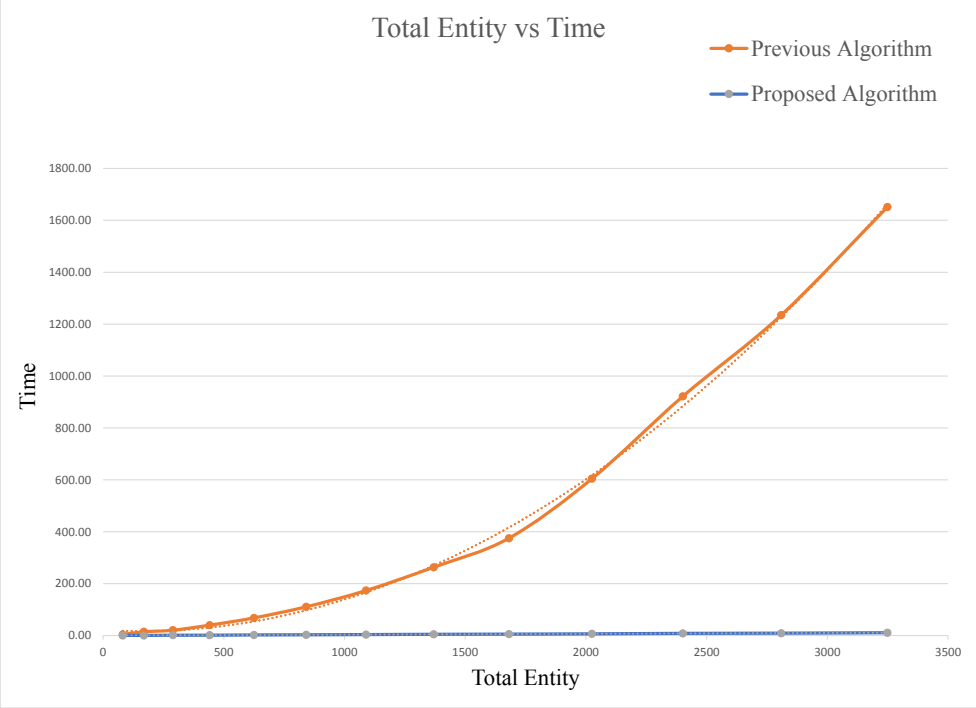


Figure 7: The time complexity for computing the conjugate of the a PLQ function approximating (5) using the algorithm from [Jak13] and the proposed algorithm .

runs in quadratic time. (We use the standard notation R^2 for the coefficient of determination that measures how close the data is to the regression line.) By contrast, a linear least-square regression fitted to the computation time of our algorithm gives $R^2 = 0.99$ thereby validating our linear computation time.

We run all numerical experiments on a Core(TM) i5 processor, 64 bit OS, 8.00 GB RAM, 2.40 GHz HP Pavilion x360 laptop, running Windows 10. The implementation of the algorithm is done using Scilab version 5.5.2. We performed the numerical experiment several times and obtained similar results each time.

The implementation of the algorithm from [Jak13] that we tested was a pure Scilab code that did not include the improvement of using the half-edge data structure provided in an external library. At the price of considerable complexity and a loss in portability, the [Jak13] algorithm can be implemented in log-linear time. However, our new algorithm would still be faster (linear-time) and much simpler.

6 Conclusion and future work

We proposed the first linear-time algorithm to compute the conjugate of a proper convex lsc bivariate PLQ function. That performance was achieved by taking advantage of adjacency information.

Further simplification was made by using graph-matrix calculus. An example provides evidence of the significant speedup of our algorithm compared to previous work.

Future work includes developing graph-matrix calculus based algorithms for computing other convex transforms like the proximal average, the Moreau envelope, the addition and the scalar multiplication for bivariate PLQ functions. The implementation for functions defined on $\mathbb{R}^d$ would also be of interest.

Acknowledgements

This work was supported in part by Discovery Grants #298145-2013 (Lucet) from NSERC, and The University of British Columbia, Okanagan campus. Part of the research was performed in the Computer-Aided Convex Analysis (CA2) laboratory funded by a Leaders Opportunity Fund (LOF, John R. Evans Leaders Fund – Funding for research infrastructure) from the Canada Foundation for Innovation (CFI) and by a British Columbia Knowledge Development Fund (BCKDF).

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows*. Prentice Hall Inc., Englewood Cliffs, NJ, 1993. Theory, algorithms, and applications.

[CGA] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[Con94] Scilab Consortium. Scilab, 1994. http://www.scilab.org.

[Fan63] K. Fan. On the Krein-Milman theorem. *Convexity*, 7:211–220, 1963.

[GJL14] Bryan Gardiner, Khan Jakee, and Yves Lucet. Computing the partial conjugate of convex piecewise linear-quadratic bivariate functions. *Comput. Optim. Appl.*, 58(1):249–272, 2014.

[GL10] Bryan Gardiner and Yves Lucet. Convex hull algorithms for piecewise linear-quadratic functions in computational convex analysis. *Set-Valued Var. Anal.*, 18(3–4):467–482, 2010.

[GL11] Bryan Gardiner and Yves Lucet. Graph-matrix calculus for computational convex analysis. In Heinz H. Bauschke, Regina S. Burachik, Patrick L. Combettes, Veit Elser, D. Russell Luke, and Henry Wolkowicz, editors, *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*, volume 49 of *Springer Optimization and Its Applications*, pages 243–259. Springer New York, 2011.

[GL13] Bryan Gardiner and Yves Lucet. Computing the conjugate of convex piecewise linear-quadratic bivariate functions. *Math. Prog.*, 139(1-2):161–184, jun 2013.

[Goe08] Rafal Goebel. Self-dual smoothing of convex and saddle functions. *J. Convex Anal.*, 15(1):179–190, 2008.

[Her16] Cristopher Hermosilla. Legendre transform and applications to finite and infinite optimization. *Set-Valued Var. Anal.*, 24(4):685–705, 2016.

[HUL93] Jean-Baptiste Hiriart-Urruty and Claude Lemaréchal. *Convex Analysis and Minimization Algorithms II*, volume 306 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 1993. Vol II: Advanced theory and bundle methods.

[Jak13] Khan Md. Kamall Jakee. Computational convex analysis using parametric quadratic programming. Master's thesis, University of British Columbia, 2013.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Series in computer-science and information processing. Addison-Wesley, 1973.

[Luc06] Yves Lucet. Fast Moreau envelope computation I: Numerical algorithms. *Numer. Algorithms*, 43(3):235–249, November 2006.

[Luc10] Yves Lucet. What shape is your conjugate? A survey of computational convex analysis and its applications. *SIAM Rev.*, 52(3):505–542, 2010.

[Luc13a] Yves Lucet. Computational Convex Analysis library, 1996-2013.

[Luc13b] Yves Lucet. Techniques and open questions in computational convex analysis. In *Computational and analytical mathematics*, volume 50 of *Springer Proc. Math. Stat.*, pages 485–500. Springer, New York, 2013.

[PS11] Panagiotis Patrinos and Haralambos Sarimveis. Convex parametric piecewise quadratic optimization: Theory and algorithms. *Automatica*, 47(8):1770 – 1777, 2011.

[PW16] C. Planiden and X. Wang. Strongly convex functions, moreau envelopes, and the generic nature of convex functions with strong minimizers. *SIAM J. Optimiz.*, 26(2):1341–1364, 2016.

[Roc70] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, 1970.

[RW98] R. T. Rockafellar and R. J.-B. Wets. *Variational Analysis*. Springer-Verlag, Berlin, 1998.